UNLIMITED

Report No. 90019

Report No. 90019

ROYAL SIGNALS AND RADAR ESTABLISHMENT, MALVERN

TRANSLATING DATA FLOW DIAGRAMS INTO Z (AND VICE VERSA)

Author: G P Randell

PROCUREMENT EXECUTIVE, MINISTRY OF DEFENCE
RSRE
Malvern, Worcestershire.

October 1990

UNLIMITED

CONDITIONS OF RELEASE

0089235 BR-115966

MR PAUL A ROBEY
DTIC
Attn:DTIC-FDAC
Cameron Station-Bldg 5
Alexandria
VA 22304 6145
USA

********** DRIC U

********** DRIC Y

Reports quoted are not necessarily available to members of the public or to commercial organisations.

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Report 90019

Title: Translating Data Flow Diagrams into Z (and vice versa)

Author: G P Randell

Date: October 1990

ABSTRACT

This report describes the results of work into the integration of structured and formal methods, in particular, data flow diagrams and Z. Data flow diagrams are given a formal, mathematical basis by defining rules to translate these diagrams into Z. Rules are also given for generating a data flow diagram from a Z specification, and for checking the compatibility of a data flow diagram and a Z specification.

THIS PAGE IS LEFT BLANK INTENTIONALLY

1. Introduction

An earlier investigation into the integration of structured and formal methods for the production of requirements specifications [1] led to the conclusion that further work is needed in all areas of the requirements definition activity. One recommendation in particular suggested that the translation of data flow diagrams into the formal language Z [2] should be researched and rules derived. This report describes the results of that research. Data flow diagrams were chosen for further study as they are in common use as a means of describing an existing or required system, but little work has been carried out in finding a mathematical basis for them. They feature in several systems analysis and design methods, including SSADM (Structured Systems Analysis and Design Method) [3], Yourdon [4] and CORE (Controlled Requirements Expression) [5], albeit in different forms.

The advantage in providing a mathematical basis for data flow diagrams is that they will become unambiguous with a defined meaning, at the same time as retaining their ease of understanding. Diagrams are often easier to understand than a mathematical specification as the latter may appear daunting at first sight. This report, by providing a mathematical basis, imposes one view of data flow diagrams, and this will be explained as the formalism is developed.

The original recommendation to translate data flow diagrams into Z has been extended and there are now three parts to the work. The first part is to produce rules for the translation of a data flow diagram into an outline Z specification, which gives a precise meaning to the diagram, and could be used in a formal development, if required. The second part is to give a formal relationship between data flow diagrams and Z specifications, which says what it means for a diagram and a Z specification to represent the same system. The final part is to give rules for generating a data flow diagram from a Z specification which may be used to illustrate and validate the specification.

This report has used the SSADM conventions for data flow diagrams, although the work is more widely applicable with some minor modifications. For example, additional constraints could be put on a diagram to incorporate desired features or a house-style.

The report is structured as follows. Section 2 formally specifies data flow diagrams in Z. This is needed before any rules can be defined. The required parts of Z are specified in Section 3, again in Z itself. The rules for translating a data flow diagram into a Z specification are given in Section 4. Section 5 describes how to check that a Z specification and a data flow diagram represent the same system. Section 6 presents the rules for generating a data flow diagram from a Z specification, and the final section contains some concluding remarks.

This report has been produced using the RSRE Z tool Zadok [7]. Zadok is a Z editor and also a syntax- and type-checker. Sections 2 to 6 of this report are separate Z documents. The conclusion of each document is marked by a "*keeps*" statement which lists the identifiers exported by the document. A document is imported into another one by including the document name, which is distinguished by having a box drawn round it.

2. The Specification of Data Flow Diagrams

In SSADM, data flow diagrams have three types of diagram element. These are **external entities**, **processes**, and **datastores**. External entities are those entities outside the system boundary from which data is input to the system, or to where data from the system is output. Processes take a set of inputs which are transformed by the process into a set of outputs. The processes carry out the functions of the system. Processes are sometimes referred to as transformers. Datastores, as their name implies, store the data held in the system.

External entities are represented graphically in SSADM by circles, processes by squares,

and datastores by two horizontal parallel lines joined on the left hand side by a vertical line. **Data flows** show data moving around the system. They are represented by arrows, labelled with the name of the data. The head of the arrow indicates the destination of the data. Each data flow joins two diagram elements together, with the constraints that a data flow cannot begin and end at the same entity, and every data flow must have a process as either its source or its destination or both.

The interpretation put on a data flow diagram in this report is that it represents operations being carried out on a state. The state is represented by the datastores, and the operations by the processes. The external entities just provide or receive data. In formalizing this interpretation in Z, specifications are needed of the relevant details of data flow diagrams and of Z. A meaning function may then be defined from data flow diagrams to Z. This function represents the interpretation.

To specify data flow diagrams, it is necessary to abstract away from the detail of the diagrams. External entities, processes and datastores will be specified first, and then the data flows between them may be specified as constraints on the set of diagram elements making up a given diagram.

Figure 1 shows an example of a data flow diagram, using the conventions of SSADM. This diagram shows part of a banking system, where a manager takes the name of a new customer and adds it to a database containing the names of the current customers. This example will be used throughout this section to illustrate the specification of data flow diagrams.

```
identifier == seq Char

external_entity
names : ℙ identifier
```

The relevant part of an external entity, from the point of view of this interpretation, is the data which it gives rise to or uses. The data moving around the system is shown by writing the name of the data, its *identifier*, on the data flow.

The example data flow diagram has one external entity, namely the customer.

```
Customer : external_entity
---------------------------
Customer.names = {"Customer_name"}
```

The data associated with the entity is just the customer's name.

Each process takes in inputs, and produces outputs, in accordance with some rules. These rules are represented by the name of the process. There are two sorts of input: those from outside of the system, that is, from external entities, and those from within the system, that is, data passed between processes directly. These are distinguished, by *inputs* and *int_inputs* respectively. Similarly, there are two sorts of output which are also distinguished.

```
process
inputs, outputs : seq identifier
int_inputs, int_outputs : seq identifier
process_name : seq Char
```

The inputs and outputs are treated as sequences rather than sets as this makes the translation rules more straightforward. The ordering can be thought of as "across the page from left to right". The sequence does not imply a particular order for processing any more than the diagram does. Flows of data between processes and datastores are not

interpreted as inputs or outputs (of any form) as they actually represent a change of the state data.

The example data flow diagram has one process, "Register_new_Customer", which has one input from an external entity, "Customer_name", and no internal inputs or any outputs. The arrow on the diagram from the process to the datastore indicates that the operation to register a new customer causes a change to the datastore, and is therefore not an output.

```
p : process
─────────────────────────────────────
p.inputs = ⟨ "Customer_name" ⟩
p.outputs = ⟨⟩
p.int_inputs = ⟨⟩
p.int_outputs = ⟨⟩
p.process_name = "Register_new_Customer"
```

The processes in a SSADM data flow diagram have two extra pieces of information associated with them, a numeric identifier and a location. The numeric identifier is unnecessary, as it is assumed that no two processes have the same name. The location does add something to the specification, if it is interpreted as describing a role. That is, states the authority needed to carry out the process. However, this information is not appropriate to incorporate into the Z specification produced by this translation.

```
datastore
─────────────────
store_name : seq Char
```

Datastores are named. The contents of stores do not appear on a data flow diagram, so no contents are specified.

The example data flow diagram has one datastore called "Customers".

```
d : datastore
─────────────────────────────
d.store_name = "Customers"
```

The identifier "D1" is not recorded as it is assumed that datastores with the same name are the same, so the identifier adds nothing.

```
DFD_element ::= ext « external_entity »
              | proc « process »
              | store « datastore »
```

The three schemas *external_entity*, *process*, and *datastore* represent the three possible type of data flow diagram element. In addition to diagram elements, a data flow diagram contains data flows:

```
data_flow
─────────────────────────────────────
origin, destination : DFD_element
label : seq Char
─────────────────────────────────────
origin ≠ destination
origin ∈ rng proc ∨ destination ∈ rng proc
```

Each data flow has a source and destination, both of which are diagram elements. Each is also labelled with the name of the data flowing along it. A data flow cannot return to its own origin, and must have a process as either its origin or destination (or both).

The example data flow diagram has two data flows, from the external customer to the process, and from the process to the datastore.

```
df1 : data_flow
---------------------------
df1.origin = ext(Customer)
df1.destination = proc(p)
df1.label = "Customer_name"
```

```
df2 : data_flow
---------------------------
df2.origin = proc(p)
df2.destination = store(d)
df2.label = ""
```

The first data flow, from the external customer to the process, is labelled with the name of the data concerned, that is, the customer's name. The second flow, however, represents a change of the state data held in the datastore, and is not labelled.

A well-formed data flow diagram consists of diagram elements connected by suitable data flows.

```
DFD
elements : 𝔽 DFD_element
flows : 𝔽 data_flow
---------------------------
∀ e : elements • ( ∃ f : flows • e = f.origin ∨ e = f.destination )
∀ f : flows • ( f.origin ∈ elements ∧ f.destination ∈ elements )
∀ e : elements; ex : external_entity | e = ext ex •
ex.names = { f : flows | e = f.origin ∨ e = f.destination • f.label }
∀ e : elements; p : process | e = proc p •
rng (p.inputs) = { f : flows
                 | f.destination = e ∧ f.origin ∈ rng ext
                 • f.label }
rng (p.outputs) = { f : flows
                  | f.origin = e ∧ f.destination ∈ rng ext
                  • f.label }
rng (p.int_inputs) = { f : flows
                     | f.destination = e ∧ f.origin ∈ rng proc
                     • f.label }
rng (p.int_outputs) = { f : flows
                      | f.origin = e ∧ f.destination ∈ rng proc
                      • f.label }
```

The first predicate says that, for each element in the diagram, there must be a flow either to or from it, or both. The second says that all flows must come from and go to a diagram element. These are basic constraints to say that the diagram must be partly connected. Additional constraints could be added to say that the diagram must contain no sources and sinks of data, or that the diagram must be completely connected (that is, is not capable of being split into two or more sets of elements, with no data flows between any two elements in different sets).

The third and fourth predicates describe two checks that must be made on each data flow diagram, to ensure that the representation in terms of elements and data flows is consistent. The first check is that, for each external entity, the names associated with it correspond to the labels on the data flows going to or coming from that entity. The second check is concerned with processes. For each process, the inputs associated with the process must correspond to the labels on the data flows from an external entity to the process, and the outputs associated with the process must correspond to the labels on the data flows from the process to an external entity. Similarly, for each process, the internal inputs associated with the process must correspond to the labels on the data flows from another process to the process, and the internal outputs associated with the process must correspond to the labels on the data flows from the process to another process.

The bank diagram is composed of one external entity, one process and one datastore, together with two data flows.

```
Bank : DFD
```

```
Bank.elements = {ext(Customer), proc(p), store(d)}
Bank.flows = {df1, df2}
```

This diagram is consistent, as the only input to the "Register_new_Customer" process is a "Customer_name", which is the same as the label on the data flow from the external Customer to the process. Also, the external Customer has only one type of data associated with it which is the same as the label on the only data flow from it. There are no data flows to the Customer.

```
Data_flow_diagram keeps identifier, external_entity, process, datastore,
                        ext, proc, store, DFD_element, data_flow, DFD,
                        Customer, p, d, df1, df2, Bank
```

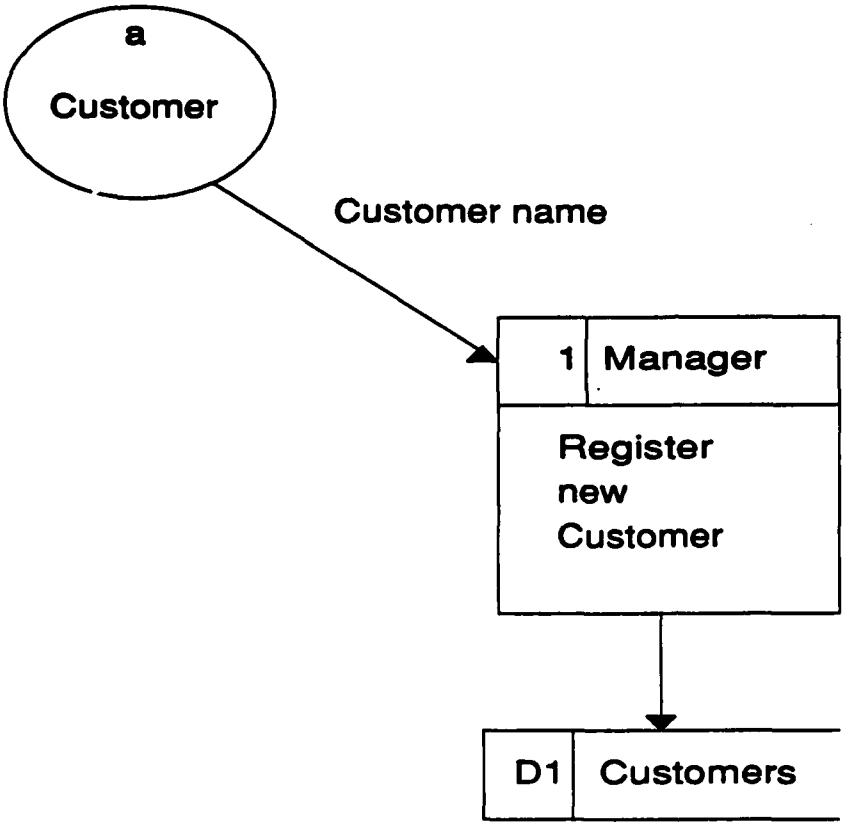


Figure 1 An Example Data Flow Diagram

3. The Specification of Z

Data_flow_diagram :Module

The purpose of this section is to specify those parts of Z that will be needed to represent any data flow diagram. For the interpretation used, only two constructs are needed, namely given sets and schemas.

A given set introduces a Z type. This type is constructed from the name of the set.

```
set == seq Char
```

The name of the set is just a sequence of characters.

Schemas are more complicated and have three parts: a name, a signature and a predicate.

```
schema_name == seq Char
```

The name of the schema is just a sequence of characters.

The signature of a schema is a more complicated structure. There are two sorts of element in a signature, inclusions and declarations. Inclusions are names of other schemas to be included, and declarations are identifier-type pairs. Identifiers have already been defined as sequences of characters. Types are more difficult.

The definitions of types and signatures are mutually recursive, so a set of types is introduced first as a given set, and then later constrained.

```
[ type ]

signature_element ::= inc « schema_name » | dec « ( identifier × type ) »

signature == ℙ signature_element
```

A signature is a set of inclusions and declarations.

```
type ::= given « seq Char »
       | tuple « seq type »
       | powerset « type »
       | schema_type « signature »
```

There are four sorts of types in Z: those constructed from given sets, tuples (which arise from cross-products), powersets (for example, *signature* above has a powerset type), and schema types. For example, the following schema:

```
example
a : ℕ; b : seq ℕ
```

has the following type:

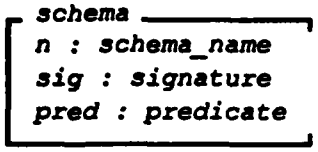
```
schema_type ⟨ dec ⟨ a, given ⟨ ℕ ⟩ ⟩,
              dec ⟨ b, tuple ⟨ given ⟨ ℕ ⟩, given ⟨ ℕ ⟩ ⟩ ⟩ ⟩
```

The predicate part of a schema is straightforward.

```
[ predicate ]

| empty : predicate
```

Nothing is said about predicates, other than that they exist, as they cannot be deduced from a data flow diagram. For the translation from data flow diagrams to Z, the empty predicate is supplied.



A schema has a name, a signature and a predicate.

```
z_element ::= given_set « set » | box « schema »
```

Each element of the part of Z used is either a given set or a schema box.

```
z_specification == ℙ z_element
```

A Z specification will consist of a set of these elements.

```
Z_specification keeps set, schema_name, signature, signature_element,
                      inc, dec, type, given, tuple, powerset,
                      schema_type, predicate, empty, schema, z_element,
                      given_set, box, z_specification
```

4. Rules for Translating a Data Flow Diagram into a Z Specification

Data_flow_diagram :Module

Z_specification :Module

This section defines a function, *translate*, which translates a data flow diagram into a Z specification. Each element of the diagram is translated into one or more Z elements: external entities are translated into given sets, processes into schemas defining operations on states, and datastores into schemas defining part of the state. A given diagram element and its translation are gathered together in the following schema along with the diagram being translated:

trans_pars

$$d\_e : DFD\_element$$
$$z\_e : \mathbb{F}\ z\_element$$
$$diagram : DFD$$

$$d\_e \in diagram.elements$$

Taking external entities first, a given set is produced for each type of data associated with that entity. These sets represent the types of the inputs and outputs to the system, and their names are given by appending "_type" to the names of the data.

trans_ext

trans_pars

$$d\_e \in rng\ ext$$
$$z\_e = given\_set\ [\!|\ \{\ n : (ext^{-1}\ d\_e).names \bullet n \frown "\_type"\ \}\ |\!]$$

Using the bank example, the external entity "Customer" produces one given set, "Customer_name_type". This set represents the only type of input to the system, and there are no outputs.

A schema is generated for each process and describes the effect the operation represented by the process has on the state of the system, that is, which datastores are affected. Additional given sets are generated for internal inputs and outputs.

```
trans_proc
trans_pars
─────────────────────────────────────────────────────────────
  d_e = proc ( θprocess )
  z_e = { box ( θschema ) } ∪
    given_set [ { i : rng int_inputs ∪ rng int_outputs • i ^ "_type" } ]
where
  | schema; process
  | ─────────────────────────────────────────────────────────
  | n = process_name
  | sig = { data_flow; datastore
  |       | θdata_flow ∈ diagram.flows
  |         store( θdatastore ) ∈ diagram.elements
  |         destination = d_e ∧ origin = store ( θdatastore )
  |         ¬( ∃ data_flow' | θdata_flow' ∈ diagram.flows
  |             • origin' = d_e ∧ destination' = origin )
  |       • inc ( "Ξ" ^ store_name ) }
  |       ∪
  |       { data_flow; datastore
  |       | θdata_flow ∈ diagram.flows
  |         store ( θdatastore ) ∈ diagram.elements
  |         origin = d_e ∧ destination = store ( θdatastore )
  |       • inc ( "Δ" ^ store_name ) }
  |       ∪
  |       { i : dom inputs
  |       • dec ( inputs i ^ "?", given ( inputs i ^ "_type" ) ) }
  |       ∪
  |       { i : dom outputs
  |       • dec ( outputs i ^ "!", given ( outputs i ^ "_type" ) ) }
  |       ∪
  |       { i : dom int_inputs
  |       • dec ( int_inputs i, given ( int_inputs i ^ "_type" ) ) }
  |       ∪
  |       { i : dom int_outputs
  |       • dec ( int_outputs i, given ( int_outputs i ^ "_type" ) ) }
  | pred = empty
```

The name of the generated schema is the same as the process name, and its signature consists of those datastores which are read by the process but not altered in any way, those datastores which the process does affect, and the inputs and outputs, both external and internal. Those datastores read but not changed are prefixed by "Ξ", while those that are changed are prefixed by "Δ". The datastores that are read but not changed by a process are found by looking for datastores in the diagram which have a data flow from them to the process, but for which there is no flow from the process back to the datastore. Each datastore which is a destination of a dataflow is changed. Datastores become inclusions in the signature.

Each input is decorated with a "?", and each output with a "!". Internal inputs and outputs are not decorated. All the inputs and outputs, both internal and external, become declarations in the signature. Their types are found by adding "_type" to the name and making the resulting name into a given set. If there were two data flows to or from the process with the same name, this name would appear only once in the signature of the process. This is a reasonable approach as it is unlikely that two outputs, say, would be identical. One may be a copy of the other, in which case it should be named accordingly as "copy_of_ ...". The given sets corresponding to external inputs and outputs are generated from external entities, while those corresponding to internal inputs and outputs are generated at the same time as the schema that uses them using this partial translation itself.

The schema is given the empty predicate. This is because information relating to the predicate does not appear on a data flow diagram but must be obtained from other sources.

The process "Register_new_Customer" from the Bank example gives rise to a schema but no additional given sets (as there are no internal inputs or internal outputs). The schema is given the same name as the process. This process affects the single "Customers" data store, and has one input, namely a customer's name.

Each datastore gives rise to a schema and a given set, which is needed to introduce the type of the contents as the form of the information held in the store is not known from the data flow diagram.

```
trans_store
trans_pars
─────────────────────────────────────────────────────────
  d_e = store ( θdatastore )
  z_e = { box ( θschema ) } ∪ { given_set ( store_name ^ "_type" ) }
where
  schema
  datastore
  ─────────────────────────────────────────────────────
  n = store_name
  sig = { dec ( store_name ^ "_contents",
                powerset ( given ( store_name ^ "_type" ) ) ) }
  pred = empty
```

The schema generated in this case has the same name as the datastore. The contents of the datastore do not appear on the diagram, so a contents list is inserted and a corresponding given set produced. This will need to be instantiated later. The contents are given a powerset type, that is, the contents of a datastore are assumed to be a set of things. As in the case of a process, the schema generated has an empty predicate.

The datastore from the Bank example produces a schema and one given set. The schema represents the state data for the system. This data consists of a set of "Customers_type".

Bringing this all together gives the following function to translate the diagram elements of a given diagram:

```
translate_element == λ d_e : DFD_element; diagram : DFD •
   ( μ z_e : 𝔽 z_element | trans_ext ∨ trans_proc ∨ trans_store • z_e )
```

The function to translate a data flow diagram into a Z specification simply translates each element of the diagram in turn:

```
translate == λ d : DFD • ⋃ { e : d.elements • translate_element (e,d) }
```

The translation of the Bank example is as follows. The external entity gives rise to one given set:

```
translate_element (ext Customer, Bank) = { given_set ("Customer_name") }
```

The process gives rise to one schema:

```
  translate_element ( proc p, Bank ) = { box (s) }
where
  s : schema
  ─────────────────────────────────────────────────────
  s.n = "Register_new_Customer"
  s.sig = { inc ( "ΔCustomers" ),
            dec ( "Customer_name?", given ( "Customer_name_type" ) ) }
  s.pred = empty
```

The datastore gives rise to one schema and one given set:

```
translate_element ( store d, Bank ) =
      { box (p), given_set ( "Customers_type" ) }
where
  p : schema
  ---------------------------------------------------------
  p.n = "Customers"
  p.sig = { dec ( "Customers_contents",
                  powerset ( given ( "Customers_type" ) ) ) }
  p.pred = empty
```

There is a total of two schemas and two given sets. When displayed as a Z specification, the result will be as follows:

```
[ Customer_name_type, Customers_type ]
```

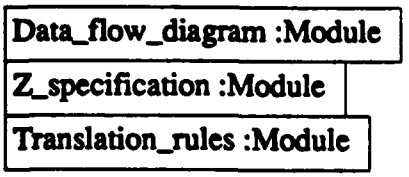
```
Customers
Customers_contents : ℙ Customers_type
```

```
Register_new_Customer
ΔCustomers
Customer_name? : Customer_name_type
```

The translation developed in this section produces a skeleton Z specification. This needs to be instantiated by defining in more detail the types of data actually used. The required information must be obtained from elsewhere as a data flow diagram cannot provide the necessary information. The translation scheme provides a useful way of using a diagram to produce a well-structured Z specification.

```
Translation_rules keeps trans_ext, trans_proc, trans_store,
                        translate_element, translate
```

5. Checking Compatibility



This section defines a relation, *represents*, which says what it means for a data flow diagram and a Z specification to represent the same system. The complete Z specification is not needed to check this compatibility. This is because a data flow diagram does not contain any information about predicates. All that is needed of the Z specification is the mapping from identifiers that appear in the specification to their types. This mapping will be called a *module_spec*. Such a mapping is produced by Zadok for each separate document (or module) of Z successfully type-checked, and forms the Z language specification of the module.

$$module\_spec : \mathbb{P}\ (identifier \pfun type)$$

The relation is defined in parts, in a similar manner to the translation function. That is, it considers each sort of diagram element in turn. A module specification is compatible with an external entity if all the names associated with that entity are in the specification; with a datastore if a schema with the same name is in the specification; and with a process if a schema with the same name and appropriate signature is in the specification.

$$\_\ partially\_represents\ \_ : DFD\_element \rel module\_spec$$

The parameters, that is, a diagram element and a module specification, are gathered into the following schema:

part_pars

$$d\_e : DFD\_element$$
$$m\_s : module\_spec$$

External entities are dealt with first. Each name associated with the external entity must be in the module specification.

part_ext

part_pars

$$d\_e \in rng\ ext$$
$$(ext^{-1}\ d\_e).names \subseteq dom\ m\_s$$

For a datastore and a module specification to be compatible there must be a schema with the same name as the datastore.

part_store

part_pars

$$d\_e \in rng\ store$$
$$(store^{-1}\ d\_e).store\_name \in dom\ m\_s$$
$$m\_s\ (\ (store^{-1}\ d\_e).store\_name\ ) \in rng\ schema\_type$$

For a process and a module specification to be compatible there must be a schema with the same name as the process and with an appropriate signature. An additional function is needed to flatten the signature of a schema, as operations are often defined in parts using several schemas and all the identifiers from all the parts need to be considered. For

example, all references to unchanged datastores may be in one schema, which is then included in the schema defining the operation. These datastores must be checked to ensure that the operation is compatible with the diagram.

```
flatten : signature → 𝔽 identifier
```

```
∀ sign : signature •
  flatten sign = { i : identifier; t : type | dec (i,t) ∈ sign • i } ∪
    ⋃ { name : schema_name; s : schema | inc (name) ∈ sign ∧ s.n = name
      • { name } ∪ flatten ( s.sig ) }
```

This function retrieves all the identifiers present in a schema signature together with all those implicitly present by virtue of a schema inclusion.

```
part_proc
part_pars
```

```
  d_e ∈ rng proc
  m_s ( (proc⁻¹ d_e).process_name ) = powerset (schema_type (sign))
where
    sign : signature; ids : 𝔽 identifier
    ids = flatten sign
    ∀ data_flow; datastore
    | destination = d_e ∧ origin = store ( θdatastore )
     ∧ ¬ ( ∃ data_flow' • origin' = d_e ∧ destination' = origin)
    • "Ξ" ⌢ store_name ∈ ids
    ∀ data_flow; datastore
    | destination = store ( θdatastore ) ∧ origin = d_e
    • "Δ" ⌢ store_name ∈ ids
    ∀ data_flow | destination = d_e ∧ origin ∈ rng ext
    • label ⌢ "?" ∈ ids
    ∀ data_flow | destination ∈ rng ext ∧ origin = d_e
    • label ⌢ "!" ∈ ids
    ∀ data_flow
    | destination = d_e ∧ origin ∈ rng proc
     ∨ destination ∈ rng proc ∧ origin = d_e
    • label ∈ ids
```

For each data flow representing a look up of state data, that is, the flow goes from a datastore to the process and there is no flow from the process back to the same datastore, the flattened signature of the schema must include "Ξstore_name". For each data flow representing a change of state data, that is, the flow goes from the process to a datastore, the flattened signature must include "Δstore_name". For each input, internal input, output and internal output, the flattened signature must contain an identifier the same as the label on the data flow, decorated with a "?" for inputs and a "!" for outputs.

Bringing this all together gives the following definition for partial representation:

```
∀ part_pars
• d_e partially_represents m_s ⇔ part_ext ∨ part_store ∨ part_proc
```

And for the overall relation,

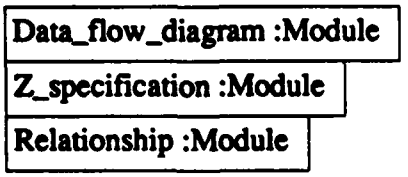
```
_ represents _ : DFD ↔ module_spec
```

```
∀ d : DFD;  m : module_spec
• d represents m ⇔
  ⋃ { d_e : d.elements; ms : module_spec
    | d_e partially_represents ms • ms } ⊆ m
```

This relation says simply that a diagram is related to a module specification if that specification contains all the sub-specifications that are related to each diagram element.

Relationship keeps module_spec, part_ext, part_store, flatten, part_proc, partially_represents, represents

6. Rules for Generating a Data Flow Diagram from a Z Specification



This section defines a function, *generate*, which generates a data flow diagram from a Z specification. A Z specification based on an example in [6] will be used to illustrate the process. This example specification is of a symbol table, with two operations defined on it: one to add an entry to the table, and one to look up the value of a symbol from the table. Two sets are introduced, representing the symbols held in the table and their values.

$[SYM, VAL]$

The symbol table itself is just a set of SYM-VAL pairs:

Symbol_Table

$$st : \mathbb{P} (SYM \times VAL)$$

The first operation is to update the symbol table with a new SYM-VAL pair:

Update

$$\Delta Symbol\_Table$$
$$sym? : SYM$$
$$val? : VAL$$

$$st' = st \oplus (sym? \mapsto val?)$$

The second operation looks up the value of a symbol in the symbol table:

LookUp

$$\Xi Symbol\_Table$$
$$sym? : SYM$$
$$val! : VAL$$

$$sym? \in dom\ (st)$$
$$val! = st\ (sym?)$$

The diagram generating function is again defined in parts, each taking a Z specification and generating part of the data flow diagram. Datastores are generated from those schemas defining (part of) the state, processes from those schemas defining operations on the state, and external entities from the inputs and outputs of the schemas defining operations. Information from the signatures of the schemas defining operations is used to generate the data flows.

One data store is generated for each schema representing part of the state.

```
gen_datastores : z_specification → ℙ datastore
```

```
∀ z : z_specification • gen_datastores z =
    { schema; datastore
    | box ( θschema ) ∈ z
      ∀ el : sig | el ∈ rng inc • hd (inc⁻¹ el) ∉ { 'Δ', 'Ξ' }
      store_name = n
    • θdatastore }
```

State schemas are identified by the absence of any schemas representing a change of state in its signature. Each datastore is given the same name as the relevant schema. Any predicate in the schema is ignored.

In the example Z specification, one datastore is generated from the schema *Symbol_Table*.

```
d : datastore
```

```
d.store_name = "Symbol_Table"
```

This datastore is given the same name as the schema.

An external entity is generated for each input and output in a schema defining an operation on the state. Each external entity generated will have one name associated with it. This could be relaxed if required so that no external entities are generated. Instead data flows representing inputs and outputs to the system could be left with one end unattached to any diagram element, and appropriate external entities added using information obtained from other sources.

```
gen_external_entities : z_specification → ℙ external_entity
```

```
∀ z : z_specification • gen_external_entities z =
   { schema; external_entity; i : identifier; ty : type
    | box ( θschema ) ∈ z
      last i ∈ { '?', '!' } ∧ dec ( i, ty ) ∈ sig
      names = { front i }
     • θexternal_entity }
```

Inputs and outputs are found by looking for identifiers ending with a "?" or "!" which appear in the signature of a schema. The type of the identifier is irrelevant, as this information does not appear on a data flow diagram.

In the example Z specification, there are two operations defined on the state, namely *Update* and *LookUp*. Each of these generates the same two external entities. Thus, the generated diagram will contain the following two external entities:

```
e1 : external_entity
```

```
e1.names = { "sym" }
```

```
e2 : external_entity
```

```
e2.names = { "val" }
```

A process is generated for each schema that defines an operation on the state. The generated process will have the same name as the schema, the inputs will be those identifiers which end with "?" in the schema, and the outputs those which end with "!". The order of the inputs and outputs does not matter. It is impossible to tell in general from the Z specification whether an identifier which does not end in "?" or "!" is an internal input or an internal output, so no attempt is made to distinguish the two. All identifiers which are not inputs, outputs or inclusions are treated as internal inputs or outputs. The predicate part of the schema is ignored.

```
gen_processes : z_specification → ℙ process
```

```
∀ z : z_specification • gen_processes z =
{ schema; process
|box ( θschema ) ∈ z
 ∃ sn : schema_name | hd sn ∈ { 'Δ', 'Ξ' } • inc ( sn ) ∈ sig
 process_name = n
 rng inputs = { i : identifier; ty : type
              | last i = '?' ∧ dec (i,ty) ∈ sig
              • front i }
 rng outputs = { i : identifier; ty : type
               | last i = '!' ∧ dec (i,ty) ∈ sig
               • front i }
 rng int_inputs ∪ rng int_outputs =
       { i : identifier; ty : type
       | last i ∉ { '?', '!' } ∧ dec (i,ty) ∈ sig
       • i }
 • θprocess }
```

A check is made that the schema does represent an operation, by requiring that there should be at least one inclusion in the signature which represents a change of state. The example Z specification generates two processes, one for each operation. These are as follows:

```
p1 : process
```

```
p1.process_name = "Update"
p1.inputs = ⟨ "sym", "val" ⟩
p1.outputs = ⟨⟩
p1.int_inputs = ⟨⟩
p1.int_outputs = ⟨⟩
```

```
p2 : process
```

```
p2.process_name = "LookUp"
p2.inputs = ⟨ "sym" ⟩
p2.outputs = ⟨ "val" ⟩
p2.int_inputs = ⟨⟩
p2.int_outputs = ⟨⟩
```

Data flows are generated for each read of the state (that is, for every "Ξdatastore" in a signature of a schema defining an operation), for each state change, and for each input and output. There is insufficient information to join up internal flows, so these cannot be filled in on the generated diagram but must be filled in by hand afterwards. This is because internal flows are just undecorated identifiers, and, if the same identifier appears in four schemas, say, there is no way of knowing which two pairs are joined by the dataflow, nor in which direction the data are flowing. The four known cases will be defined in turn.

Each read of the state generates a flow from the datastore to the process concerned.

```
gen_data_flows_1 == λ z : z_specification •
   { schema; data_flow; i : identifier
   | inc i ∈ sig ∧ hd i = 'Ξ'
     origin = store ( μ ds : (gen_datastores z)
                    | ds.store_name = tl i • ds )
     destination = proc ( μ p : (gen_processes z)
                        | p.process_name = n • p )
     label = ""
   • θdata_flow }
```

Reads are identified by the presence of an inclusion of the form "Ξschema_name". The origin of the flow is the store with the corresponding name, and the destination the process with the same name as the schema in which the inclusion was found. No label is put on the data flow.

Each state change generates a flow from the process concerned to the relevant data store.

```
gen_data_flows_2 == λ z : z_specification •
   { schema; i : identifier; data_flow
   | inc i ∈ sig ∧ hd i = 'Δ'
     destination = store ( μ ds : (gen_datastores z)
                           | ds.store_name = tl i • ds )
     origin = proc ( μ p : (gen_processes z)
                     | p.process_name = n • p )
     label = ""
   • θdata_flow }
```

State changes are identified by the presence of an inclusion of the form "Δschema_name". The origin of the flow is the process with the same name as the schema in which the inclusion was found and the destination the store with the corresponding name. No label is put on the data flow.

Data flows are generated for each input:

```
gen_data_flows_3 == λ z : z_specification •
   { schema; i : identifier; data_flow
   | i ∈ dom ( dec⁻¹ ⦇ sig ⦈ ) ∧ last i = '?'
     origin = ext ( μ e : (gen_external_entities z)
                  | e.names = {front i} • e )
     destination = proc ( μ p : (gen_processes z)
                          | p.process_name = n • p )
     label = front i
   • θdata_flow }
```

Inputs are identified by the existence of a declaration in a schema, with identifier ending with a "?". The origin of the flow is the external entity with the identifier without the "?" as its associated name, which is also the label put on the data flow. The destination is the process with the same name as the schema in which the input was found.

Data flows are generated for each output:

```
gen_data_flows_4 == λ z : z_specification •
   { schema; i : identifier; data_flow
   | i ∈ dom ( dec⁻¹ ⦇ sig ⦈ ) ∧ last i = '!'
     destination = ext ( μ e : (gen_external_entities z)
                         | e.names = {front i} • e )
     origin = proc ( μ p : (gen_processes z)
                     | p.process_name = n • p )
     label = front i
   • θdata_flow }
```

Outputs are identified by the existence of a declaration in a schema, with identifier ending with a "!". The destination of the flow is the external entity with the identifier without the "!" as its associated name, which is also the label put on the data flow. The origin is the process with the same name as the schema in which the output was found.

Bringing this all together into one rule gives:

```
gen_data_flows : z_specification → ℙ data_flow
─────────────────────────────────────────────
∀ z : z_specification
• gen_data_flows z =  ( gen_data_flows_1 z )
                    ∪ ( gen_data_flows_2 z )
                    ∪ ( gen_data_flows_3 z )
                    ∪ ( gen_data_flows_4 z )
```

The example Z specification generates six data flows. The operation `Update` generates one from the update process to the datastore. It also generates one from the external entity with name "sym", and one from the external entity with name "val". The operation

LookUp generates one data flow from the datastore to the lookup process, one from the external entity with name "sym" and one to the external entity with name "val".

Bringing all the partial generating functions together gives the following function for generating a data flow diagram from a Z specification, provided that the specification is written in the style of state schemas and operations on (part of) the state:

```
generate : z_specification ⇸ DFD
─────────────────────────────────────────────
∀ z : z_specification •
  generate z = d
where
  d : DFD
  ──────────────────────────────────────────
  d.elements = store ( gen_datastores z )
             ∪ ext ( gen_external_entities z )
             ∪ proc ( gen_processes z )
  d.flows = gen_data_flows z
```

The diagram resulting from the example developed in this section is shown in Figure 2.

Translation rules have been developed not only for developing a Z specification from a data flow diagram (in Section 4), but also for generating a data flow diagram from a Z specification, described in this section. This latter is a useful way of illustrating a Z specification so that the structure becomes clear. The diagram produced will aid understanding of the specification and will help with the process of validating the specification.

```
Generating_rules keeps gen_datastores, gen_external_entities,
                       gen_processes, gen_data_flows, generate
```

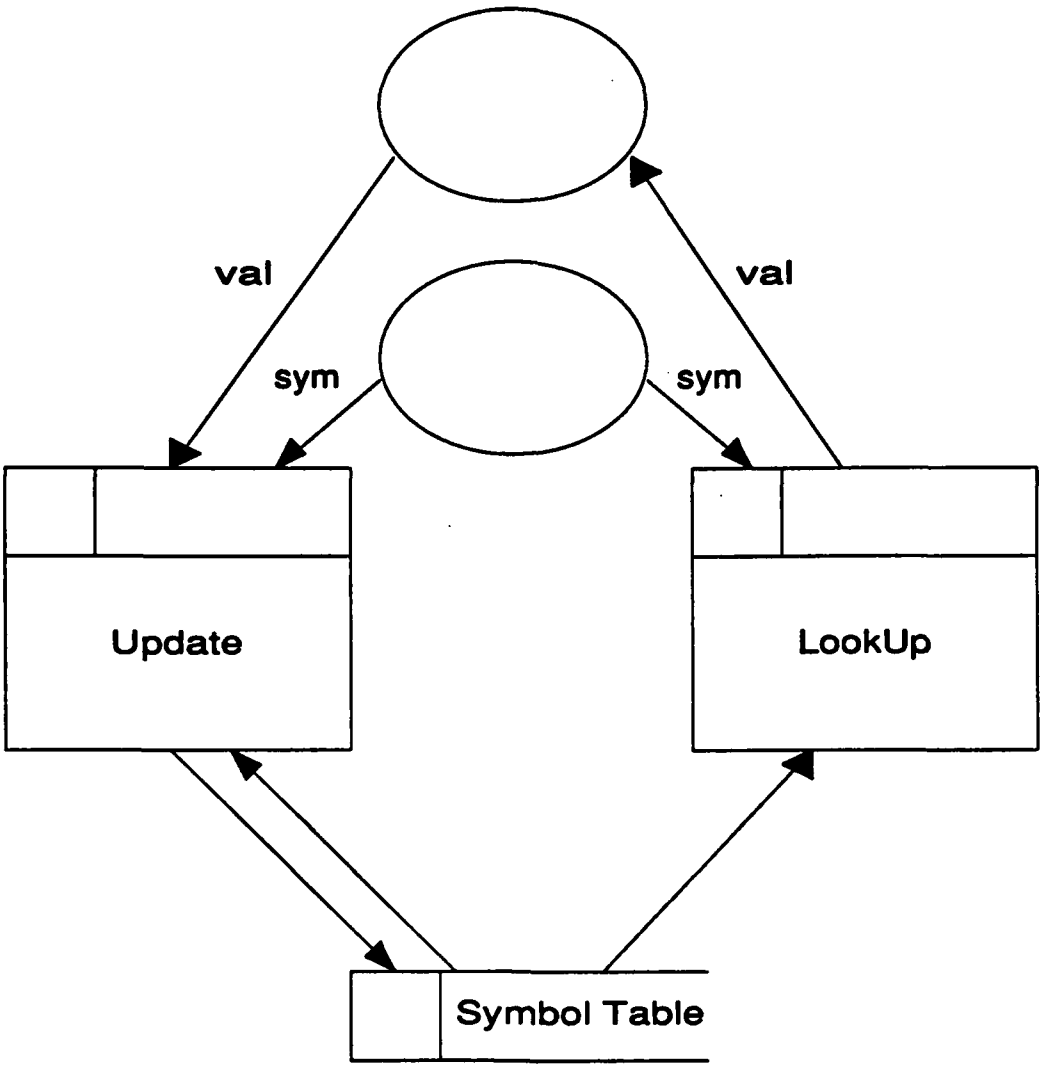


Figure 2 The Generated Data Flow Diagram

7. Conclusions

This report has presented three approaches to using a formal language, Z, with a structured diagrammatic technique, data flow diagrams. These approaches are translating data flow diagrams into a Z specification, checking consistency of a diagram and a Z specification using the relationship between them, and generating a diagram from a Z specification written in a state-based style.

A particular interpretation has been put on both data flow diagrams and Z, and this needs to be validated. However, it is based on the standard style of using Z, that is, using schemas to specify both the state and operations on that state. Translating a data flow diagram into Z produces only an outline specification. This specification should be used as a starting point, a way of structuring a specification. Conversely, generating a data flow diagram from a Z specification may lead to a large diagram. This would occur in the case where an operation is defined in parts, each part in a separate schema. If the complete operation only were used in the generation process then a sensible diagram would result. If, however, diagram elements were generated for each part, then the resulting diagram would be cluttered. A useful approach would be to use the separate parts of an operation to generate a separate diagram describing just that operation.

The data flow diagram may be considered to be a high level overview of the system. Therefore, it is not surprising that a small amount of Z is produced. The usefulness of the relationship defined is that it provides a mechanical check that the high level overview given by a data flow diagram is consistent with the more detailed description given by a Z specification. There are other information flows contained in a Z specification, for example, the inclusions of schemas within other schemas. A similar diagram to a data flow diagram could be automatically generated to produce a "usage tree" of the specification. This would be of great help to reviewers of the specification as it would help them to navigate through the specification and keep track of the usage of schemas.

The specifications given in this report could be used to build a tool to carry out the three techniques, preferably as part of an integrated support tool. However, data flow diagrams are only one of a number of techniques used in a structured method such as SSADM. Other diagrammatic techniques include entity life histories and logical data structuring (also called entity modelling). This work with data flow diagrams and Z has demonstrated the usefulness and practicality of combining structured and formal methods, and a follow-on to this would be to carry out similar studies for the other diagrammatic techniques. The eventual aim is a toolset which will allow outline formal specifications to be generated automatically from a set of diagrams, which may then be used in a formal development, if desired, of the required system. Alternatively, the mathematical specifications may be used to prove properties of the system. Generating diagrams from a formal, mathematical specification will aid validation of that specification as diagrams are easier to understand than a mathematical specifiation, and are not so daunting at first sight.

This work, by providing a set of formal rules to translate a diagram into a formal language, has resulted in the diagrams themselves becoming formal objects, with all the associated benefits of precision and lack of ambiguity. Adding this formal basis has not detracted from the readability of the diagrams. This demonstrates that formal languages are not necessarily harder to understand. It is the mathematical symbols of current formal languages such as Z that make them seem more daunting than they really are.

References

[1] G P Randell, "The Integration of Structured and Formal Methods", RSRE Memorandum No. 4293, June 1989

[2] J M Spivey, "The Z Notation: A Reference Manual", Prentice-Hall International Series in Computer Science, 1989

[3] G Longworth and D Nicholls, "SSADM Manual", Version 3.0, NCC, December 1986

[4] Yourdon International Ltd., "The Yourdon Structured Method (YSM)", Yourdon, 1988

[5] Systems Designers Scientific Ltd., "CORE - the Method", Issue 1.0, November 1985

[6] I Hayes (Editor), "Specification Case Studies", Prentice-Hall International Series in Computer Science, 1987

[7] G P Randell, "Zadok User Guide", RSRE Memorandum No. 4356, January 1990

THIS PAGE IS LEFT BLANK INTENTIONALLY

REPORT DOCUMENTATION PAGE

DRIC Reference Number (if known) .....

Overall security classification of sheet .....UNCLASSIFIED.....
(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the field concerned must be marked to indicate the classification eg (R), (C) or (S).

Originators Reference/Report No.	Month	Year
REPORT 90019	OCTOBER	1990

Originators Name and Location
RSRE, St Andrews Road
Malvern, Worcs WR14 3PS

Monitoring Agency Name and Location

Title
TRANSLATING DATA FLOW DIAGRAMS INTO Z
(AND VICE VERSA)

Report Security Classification	Title Classification (U, R, C or S)
UNCLASSIFIED	U

Foreign Language Title (in the case of translations)

Conference Details

Agency Reference	Contract Number and Period

Project Number	Other References

Authors	Pagination and Ref
RANDELL, G P	23

Abstract

This report describes the results of work into the integration of structured and formal methods, in particular, data flow diagrams and Z. Data flow diagrams are given a formal, mathematical basis by defining rules to translate these diagrams into Z. Rules are also given for generating a data flow diagram from a Z specification, and for checking the compatibility of a data flow diagram and a Z specification.

Abstract Classification (U,R,C or S)
U

Descriptors

Distribution Statement (Enter any limitations on the distribution of the document)
UNLIMITED

ssss

THIS PAGE IS LEFT BLANK INTENTIONALLY